AD-A225 812

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	April, 1990	Final Report, 30 Sep 87 to 29 Jun 90

4. TITLE AND SUBTITLE
APPLICATIONS OF MASSIVE MATHEMATICALCOMPUTATIONS

5. FUNDING NUMBERS
F49620-87-C-0113
61101E 6120/D0

6. AUTHOR(S)
David V. Chudnovsky

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Columbia University
Department of Mathematics
New York, NY 10027

8. PERFORMING ORGANIZATION REPORT NUMBER
AFOSR-TR 90 0862

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFOSR/NM
Bldg 410
Bolling AFB DC 20332-6448

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release;
distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

During the period of the grant our group has been using super-computers and parallel machines to solve important, large and realistic problems. A distinctive characteristic of these problems, apart from their sheer size, is that they required the development of new algorithms which will be useful for other mathematical and physical problems, as well as for improvement of computer performance and reliability.

14. SUBJECT TERMS	15. NUMBER OF PAGES
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR

Re: Contract No. F49620-87-C-0113

Final Technical Report

to

D. A. R. P. A.

Applications of Massive Mathematical Computations

Columbia University

April, 1990

7/2/90

Summary

During the period of the grant our group has been using supercomputers and parallel machines to solve important,large and realistic problems. A distinctive characteristic of these problems, apart from their sheer size, is that they required the development of new algorithms which will be useful for other mathematical and physical problems, as well as for improvement of computer performance and reliability.

1. Multifluid code.

Milestones.

This code, the result of years of development and testing, realized its full potential when implemented in a parallel environment. The multifluid code in two and three dimensions (plus time) is currently running on the IBM GF11 parallel supercomputer – this machine has 576 processors, and reaches has a peak performance of 11.2 gigaflops. To the best of our knowledge, we are the first group to achieve stable, long-term (30,000 step) multifluid computations on a 100 X 100 X 100 grid, including chemical reactions and associated energy release. An efficiency of more than 70% of the GF11's performance was realized in these computations.

Technology Transfer:

Quite aside from the general scientific utility of a complex multifluid code, our work serves an important purpose in the day-to-day operation of the GF11: as an operating code which offers a comprehensive test of the 576 processors and interconnects it is routinely used for diagnostics of the machine. Another important part of the multifluid code is its fast special function package, which was generated with the help of computer algebra.

2. Special Function Facility.

Milestones.

The principal investigators applied their new algorithms of symbolic power series manipulations to construct a comprehensive and efficient facility for evaluation of and computations with classical and user - defined special functions. This facility allows the user to achieve arbitrary precision in the calculation of a special functions everywhere on the appropriate Riemann surface. Such a comprehensive facility is of great demand in many applications. To mention a few:

1) in medicine, Radon transforms for CAT scans;

2) in theoretical physics, the special functions that arise in studies of atomic, molecular and solid state problems;

3) in nuclear physics, neutron diffusion;

4) in astrophysics, radiative transfer;

5) in mathematics, modular functions and number theory.

This is the first time that such a comprehensive facility has been developed in full generality, and also the first time that it was implemented in arbirary precision.

Technology transfer.

The symbolic manipulation program described above will be available (with a user-friendly interface) in SCRATCHPAD (IBM) in 1990.

3. π and the solution of very large linear and nonlinear problems.

Milestones.

The most challenging test of supercomputer performance is an error-free computation that pushes the limits of storage and requires weeks of CPU time. The computation of π is traditionally just such a test of computer reliability. Numerical demands in the computation of π are severe: to compute N decimal digits of π with the best algorithm

developed by the principal investigators, requires a large number of convolutions (integer Fourier transforms, etc.) of size O(N). The principal investigators have designed efficient self-verifying algorithms for solving the large number of the numerical problems needed for such tasks: convolutions of large arrays, polynomial and integer multiplications of lengths in gigabytes, solution of linear algebra problems involving matrices with hundreds of millions of elements, etc. Our successful computation of more than one billion decimal digits of π considerably surpasses the competing Japanese supercomputer effort using Hitachi 820/80 and NEC SX supercomputers. Our computations were performed independently on CRAY-2 and IBM 3090 machines. What might have been lost in a reading of the nontechnical publications about this effort is the importance of the algorithms, techniques and codes developed by the principal investigators in the course of computing π. For the first time a large problem that does not fit into a supercomputer's physical memory was solved without loss of elapsed time. In our computation of π, for example, we routinely perform 256 megaword (=2 gigabytes) of error-free convolution on data residing externally on disks, or even on tapes. The potential of this technique is difficult to overestimate, because it bypasses a major roadblock in the efficient use of supercomputers: the size of the physical memory of a supercomputer has often set an effective limit to the size of the problem which it could efficiently solve.

Technology transfer.

The techniques described above are already in use: the size of linear algebra problems that can be solved on an IBM 3090 are limited only by the external storage capacity. The run time for the solution of a dense 17K X 17K ($17K = 17 \cdot 2^{13}$) system of linear equations externally stored is 8.7 hours on 1 processor (out of 12), and requires no large amount of physical memory.

4. Truly Parallel Algorithms for the Finite Element Method.

Milestones.

The Finite Element Method is the most widely used computational tool in industry for the solution of complex technological problems. For the highest efficiency to be realized on a parallel computer it is essential that the algorithms be asynchronous in character, i.e. they must be decomposable into sets of processes with minimal explicit interdependence. Algorithms designed for serial computers do not have this property; our newly-created algorithms do. Moreover, they are designed to operate with any number of independently programmable processors.

We have implemented highly efficient asynchronous algorithms on the memory-sharing multiprocessor Balance Computer and on IBM 3090 multiprocessor supercomputers. We have established experimentally that the new algorithms produce almost perfect speed-up on large-scale problems.

Multi Fluid Flows With Chemical Reactions and Thermodynamical Effects. GF11 Applications.

In this section we present the description of our multiflud flow parallel codes in 2D and 3D, and their implementation on IBM GF11 parallel supercomputer. We used our codes to simulate galactic formation and evolution for a variety of initial conditions and physical parameters. Problems encountered include optimization of the code for machine execution, particularly: issues of communication, minimization of the computational complexity of the code, increasing the efficiency of the code for specific supercomputers, and representation and visualization of resulting data. The mathematical part of the development involved the generation of efficient algorithms for special function evaluation and numerical quadrature, assisted by computer algebra systems, as well as (stability) analysis of preservation of isothermal solutions. The environment for supercalculation on GF11 is described.

The problem of the hydrodynamics of coexisting fluids is a problem of great contemporary interest. It has immediate technological applications particularly for the study of the aerotheromodynamics of hypersonic aircraft and the reacting gases of the ramjet or rocket engines that will power them. The complexity of the problem chosen far exceeds that of conventional fluid dynamics computations. The particular examples computed were based upon a problem from astrophysics, the simulation of the formation and evolution of galaxies, which have the same fundamental complexity but provide a more convenient data base for study the efficiency of algorithms. In our study, chemical reactions within the first fluid produce the second fluid: the second fluid provides a heat source for the first one, which would otherwise cool by itself. The first of these fluids we identify with

interstallar matter, and the second with stars.

Supercalculations on GF11.

The main characterization of supercalculations is the amount of number crunching required to solve a particular problem. As an example of a working definition, one can consider a calculation to be super, if it takes more than a trillion (one Teraflop) of calculations (floating or fixed point) to finish the job. Supercalculations are those that one cannot finish in a week of VAX time.

Supercalculations often result when large amounts of data must be processed. This is particularly true in grid computations that represent a finite element or a finite difference realization of a continuous physical model. The amount of data recalculated in these models as time advances is determined by the total number of "cells" from which the grid structure or finite elements are built. To approximate a continuous physical model, particularly in 2D or 3D analysis, the grid is often very large. It now not uncommon to run computations for grids with dimensions of the order 100 x 100 x 100 to 256 x 256 x 256, even for scientists having no access to large amounts of supercomputer time.

The supercalculations discussed in this paper were suggested by K. Prendergast. These are large scale (both in data volume and in run time) 2D and 3D astrophysical simulation of galactic dynamics. Codes developed for this purpose are (three-dimensional) multi-fluid codes, that are first order, explicit, flux splitting, and conserve mass, momentum and energy (even in the presence of gravitational forces).

This Galaxy project is designed to construct computer models of galaxies which embody the interaction of stars and gas in our own galaxy, and which agree with the observed forms, colors, light distributions and internal motions of other galaxies. The number of arithmetic operations per single test case (and we plan to accumulate thousands of them) is quite large: it varies from about 60 billion for

small 2D problems to 30 trillion for larger 3D modeling.

The first runs involve modeling of two-dimensional star and gas interactions (without gravity) on hundreds of different cases. The runs differ in the laws governing star formation and gas cooling. Each case will show the physical processes at work in a representative piece of interstellar matter, under a particular set of physical assumptions, during a physical time corresponding to a substantial part (roughly a quarter) of the age of the Universe. The purpose of this modeling is to settle on the "best" parametrization of star formation and gas cooling in the simulation of an entire galaxy (with self gravitation). Test case runs so far show the growth of a foam-like structure comprising cold clouds, filaments and hot bubbles in the simulated interstellar medium. The whole structure evolves in time - stars are formed in the clouds, heat and disrupt them, and eventually produce a bubble of hot gas. The gas in each bubble expands until it impinges on another bubble, and a cool, dense filament forms along their common boundary. Cold clouds form at the intersections of filaments and grow as gas drains along the filaments. Clouds continue to grow until enough stars have formed to disrupt them, and the cycle starts over. Other important components of the three-dimensional code include a parallel Poisson solver (to find the gravitational field, given the density) developed by R. Miller.

The particular supercomputer used for these calculations is the IBM GF11 experimental parallel machine, described below. Code development was targeted for this machine, but at the same time the structure of the code was adopted to suit a large class of modern vector and parallel supercomputers. To compare results and performance, runs of the Galaxy code were carried out on a variety of more conventional machines.

One of the consequences of calculation is the need to interpret and display a vast amount of data representing galactic simulations. For this purpose several graphics programs were created to visualize the dynamics of galactic evolution in space and time. One of the fastest methods in practice turned out to be a display of random sampling of

the mass (density) distribution of stars and gas (interstellar matter) with proper color coding. This representation allows one to achieve a near-animated effect for a 2D time evolution with modest means and minimal storage requirements. In 3D case, animation effects are possible only with a modest sampling displayed on a computer screen (several thousand of sample points). Conventional methods of visualizations of 2D data (landscapes, etc.), and new means of graphic representation of 3D data were implemented as well.

Successful realization of Galaxy astrophysical computations on GF11 was accomplished due to an efficient code generator for this machine, which turned out to be satisfactory for the development of a structured scientific code on GF11 in a high-level language form, even in the absence of a conventional FORTRAN complier. Other examples of successful supercalculations performed on GF11 are also described below. Programming efforts were a joint effort with IBM Research, particularly with M. Denneau and Y. Baransky.

1. Basics of the Code.

The basic hydrodynamical code was constructed by exploiting the well-known fact that the equations of hydrodynamics are moment equations of the Boltzmann equation

$$Df/Dt = (df/dt)_{coll}$$

When collisions are frequent, the (mass) distribution function f is never very different from a Maxwellian, and consequently both sides of the equation are in some sense small, except in shocks or contact discontinuities. We can construct a useful hydrocode by making the right and left sides alternately zero. Suppose we make $Df/Dt = 0$ for a short time τ; this implies that in the absence of external forces

$$\mathbf{f}[\mathbf{r} - \mathbf{u}\tau, \mathbf{u}, t + \tau] = \mathbf{f}[\mathbf{r}, \mathbf{u}, t],$$

i.e. each particle moves in a straight line of constant velocity. If at time t we start with a uniform density of mass, momentum and energy within each cell of the grid, we can construct a (unique) uniform Maxwellian distribution function $f^M[\mathbf{r}, \mathbf{u}, t]$ having these densities f_i^M for each cell. The flux of any quantity Q from one cell [i] to a neighboring cell [i+1] across the boundary separating them is

$$\int f_i^M \, Q \cdot \mathbf{u} \cdot \mathbf{n} d\mathbf{u}.$$

By letting $Q = 1, \mathbf{u}$ and $\mathbf{u}^2/2 + h$, where h is the internal energy of a molecule we can find the flux of mass, energy and momentum from [i] to [i+1]. Analogous fluxes of these quantities from [i+1] to [i] are computed using f_{i+1}^M. The total change in mass, momentum and energy in each cell in time t can therefore be computed from the geometry of the cells and $f_i^M(t)$. Of course, during the time interval τ the distribution function changes from the original Maxwellian form so that $|df/dt|_{coll}$, originally zero, is (generally) a nondecreasing function of τ. Since the collision term must be small in the hydrodynamic limit, we restart by recomputing the parameters of the new Maxwellian f_i^M at time $t + \tau$ from the new values of mass, momentum and energy in each cell. In practice the time step τ is chosen to satisfy a Courant-Lewy-Friedrichs condition. The flux integrals for an exact Maxwellian f^M involve error functions, which can be expensive to compute, and are not necessary for many problems. The performance of the code does not seem to be adversely affected if f^M is replaced by a function g^M having the same first three moments with respect to u as f^M. We have used at various times weighted sums of three and four delta-functions, as well as a parabolic approximations in u for g^M. More general higher order polynomial "entropy" approximations $P_n(x)$ are described below.

If we adopt δ-functions for g^M, we get a code closely related to the so-called "beam scheme", which is straightforward and efficient on scalar machines, particularly for one- and two- dimensional com-

putations. However, a major drawback of this scheme for large scale computations is the need for large number of scratch working arrays - for the problems we studied one needs 5 extra arrays per each data array (5 beams) in 2D and 6 scratch arrays per each data array in 3D. This amount of storage is unbearable even for large machine, [e.g. in 2 fluid computations one needs at least 10 input /output data per cell - 6 beams make the amount needed for physical storage unbearable for a 200 x 200 x 200 grid].

The total amount of computations with the beam scheme is fairly large. We counted the number of arithmetic operations in the 2D case. (Not to be confused with cycle count in actual execution on any machine):

To update the density, momentum vector, and energy using a beam scheme one needs

154 multiplications

131 additions

2 divisions

and 1 *square root*

per cell. Usable grids for galactic simulations start at 100 x 100 in 2D. In problems we were interested in the number of discrete time steps needed varied between 104 to 105.

In the 3D case the beam scheme is inefficient because of the increase in storage and number of operations per cell -the number of data per cell increases because of dimension, and so does the number of beams.

The worst problem with the beam scheme is an inherently unparallelizable (or nonvectorizable) feature in the method. In this scheme flux transfer occurs in a single direction to any of 8 neighbor cells in the 2D case, and to 26 neighbor cells in the 3D case. As a result, the direction of transfer varies from cell to cell and moreover, a single cell can receive data update from each of 26 directions. Thus the process of updating is unsynchronized, and greatly suffers in performance on a vector hardware or any SIMD machine (a possible hit can be by a factor of 4 in the 2D case and 14 in the 3D case). Another problem of

crucial importance for a vector machine is the dominance of load/store operations in the beam scheme versus floating point operations, which prevents high efficiency execution on existing supercomputers.

The introduction of gravitational forces requires non-trivial modification of the code, for several reasons. The first is that we require exact conservation of energy, now including gravitational energy. The second is that the solution of Df/Dt = 0 is now non-trivial, as it requires a knowledge of the trajectory of a particle in a potential field for arbitrary initial conditions. The third complication is a matter of some subtlety - it turns out that our freedom to choose a convenient substitute for the exact Maxwellian is completely lost when gravitation is included in the code.

In our treatment of gravitational forces we have made the somewhat brutal assumption that the potential is constant in a cell - this enables us to retain the constancy of mass, momentum and energy densities (or equivalently, of the parameters of the Maxwellian) within a cell. The motion of a particle in such a potential can found explicitly and therefore it is possible to conserve total energy "exactly". The flux of momentum across a wall between two cells with different potentials cannot be computed in terms of classical functions for a Maxwellian. It is here that we encountered the two integrals Y_1 and Y_2 (see next section), which are needed to compute the momentum flux in a gravitational field.

It might be thought that one could completely avoid the use of Y_1 and Y_2Y1 entirely by using a different distribution function (perhaps a maximum entropy polynomial) but this is not true. To see this, consider the class of test cases furnished by the well-known result from Statistical Mechanics that the distribution function in thermodynamic equilibrium is

$$f = A\exp(-\beta E) = A\exp(-\beta m[|\mathbf{u}|^2/2 + \Psi]),$$

with A, β constant throughout the system. Consider for simplicity a one dimensional problem for given $\Psi(x)$ and with periodic boundary

conditions. It is easy to show that the conservation of mass leads to a functional equation for the distribution function employed in the computation, and that this equation must be independent of the potential. The solution of this functional equation is a Maxwellian - that is, only the exact Maxwellian distribution function can be used in the numerical scheme if we insist that the code accept the isothermal solution for an arbitrary potential.

We expect that an analogous result may hold for a number of otherwise plausible codes if there is a gravitational field. Even so, the consequences may be less devastating than would appear at first sight, since most problems are not run to thermal equilibrium.

2. Special Functions Evaluations in Astrophysical Codes.

Large, robust, 3D fluid dynamics, plasma physics and astrophysics codes sometimes have high computational complexity per cell, because of special and elementary function evaluations needed for recomputation at each discrete time step. While elementary function evaluation is often efficiently performed by a supercomputer (e.g., IBM or CRAY specialized scientific libraries), it is inevitably slow compared to basic arithmetic operations. [Division or taking a reciprocal has to be considered on most supercomputers as an elementary nonprimitive and slow operation]. Computations of special functions, such as the incomplete Gamma-function, the error function, Bessel functions, etc. needed in many supercalculations are usually based on standard polynomial or rational function approximation (typically uniform approximations) schemes. These computations tend to dominate many specialized codes.

In the code we were interested in, nonprimitive operations include large number of divisions, square roots, exponents, trigonometric functions, error functions and integrals of their combinations. The only way to avoid a large number of special function evaluations is to change the physical model. One such remedy, presented

above, is the beam scheme, in which the Gaussian distribution is replaced by a sum of delta-functions. While this is quite satisfactory in aero/hydrodynamics computations, gravitational effects (or effects of other potential fields) cannot be simulated with this approximation, as explained above. Moreover, the operation count in the 3D beam scheme is actually larger than in a 3D code using error functions. Since special functions are needed, efficient methods of their computations have to be used. Many of these special functions (like integrals in the gravitational codes) depend on several parameters, with markedly different behavior in different regions. Traditionally such integrals were evaluated using numerical integration techniques and optimized packages. This made large runs of codes unfeasible, even on supercomputers. We touch upon novel techniques of special function evaluation for problems in astrophysics and physics, based on code development using computer algebra systems, particularly SCRATCHPAD (IBM).

First, a careful work was conducted in SCRATCHPAD to lower the algebraic complexity of the polynomial and rational part of Galaxy code. Let us give the basic count for chemistry / thermodynamics / flux transfer part of the 2-fluid 3D code. In this code one considers a general grid of the size Mx x My x Mz.

Updated physical data representing the state of the galaxy are:
density (mass),
momentum vector (3 components),
and energy (or temperature)
for each grid element (cell) and for each kind of matter: gas and stars. If dark matter is included, there are 3 sets of such data per cell.

During each discrete time step all these data are recomputed. Chemistry / thermodynamics and flux transfer require:
228 additions,
192 multiplications,
8 divisions,
2 square roots,
6 exponents,

and 6 error function computations on single precision floats per each of Mx x My x Mz cells of the grid.

For serious purposes, other than debugging, each of the linear dimensions has to be at least 100 (values between 100 and 200 seem to be optimal both from computational and astrophysical points of view).

The run time can be estimated as 104 to 106 discrete time steps - iterations (in fact, only the a priory physical time in the evolution of galaxy matters, this time, however, cannot be a priory bounded, because the Courant condition establishes a nontrivial expression for one unit of physical time in terms of a number of discrete time steps.

The complexity above does not take into consideration load, store and interprocessor communication time. The memory interfaces are very heavy and 18 arrays, each of the total grid size are accessed and overwritten multiple times during the recalculation steps.

Our code currently is totally vectorized for single processor execution. For multiprocessor run, the chemical and thermodynamic parts of the code are split. Flux transfer interprocessor communication is slightly more involved and uses, in case of p riodic boundary conditions, modular arithmetic to represent tori as one dimensional arrays. When self-gravitation and an n-body problem are added, interprocessor communication gets more involved, and, in particular, 3D convolutions are performed on data spread across the processors.

All versions of the Galaxy code, including the spherical code, developed by Prendergast and Athanassoulas, require frequent evaluation of exponential and error functions.

The error function referred to above is defined as:

$$\mathrm{erf}(x) = 2/\sqrt{\pi} \int_0^x \exp(-x^2)dx.$$

High precision method of evaluation of erf(x), known for many years, were highly popularized by Henrici in his numerous papers and books. Henrici divides the positive x-axis into 2 regions: x ¡ 1.5 and x =¿ 1.5; near the origin one uses Kummer's identity for

the hypergeometric ${}_1F_1$ function and evaluates erf(x) from the power series expansion at the origin and the representation

$$erf(x) = 2(x/\sqrt{\pi} \cdot \exp(-x^2) \cdot {}_1F_1(1; 3/2; x^2).$$

For larger x, one uses the complementary, erfc-function and converts its asymptotic series at infinity:

$$erfc(x) = 1 - erf(x) = \exp(-x^2)/(x\sqrt{\pi}) \cdot {}_2F_0(1/2, 1; -1/x^2)$$

into a Gauss continued fraction expansion:

$$erfc(x) = \exp(-x^2)/(1\_|^{-}x + 1/2|^{-}x + 1|^{-}x + 3/2|^{-}x + 2|^{-}x).$$

For single precision computations (say, with a 24-bit mantissa), a simpler scheme based on *uniform polynomial approximation* is well known.

In the presence of a gravitational field, all local computations of flux transfer, in addition to evaluation of exponents and error function we require the evaluation of two integrals depending on two parameters - the velocity and the local gravitational force. We express these two integrals in the following form:

$$Y_1(a|c) = \int_0^\infty \exp(-(x-c)^2) \cdot x \cdot \sqrt{(x^2 - a^2)}\, dx,$$

$$Y_2(c|d) = \int_0^\infty \exp(-(x-d)^2) \cdot x \cdot \sqrt{(x^2 + c^2)}\, dx,$$

In the 3D code, per cell at each discrete time step, one has to evaluate such integrals 12 times. By far, this is the most number crunching intensive part of the Galaxy code. Sadly, library subroutines based

on Hermite's numerical quadratures available for vector machines are unsatisfactory for our integrands because of special behavior at the end and, often, in the middle of the interval of integration. Traditional Romberg gives satisfactory results, but requires a large number of evaluations of the integrand, particularly when there is a need to increase the precision of calculations.

Studying the functions defined by the integrals above in SCRATCHPAD, we found that they can be much more efficiently evaluated from their power series expansions, not unlike error functions. These integrals are related to hypergeometric integrals, and one can study them as solutions of linear differential equations. In the complex plane the two integrals above represent branches of the same function. These integrals satisfy a system of linear differential equations in both parameters. Particularly interesting is an equation in one of these parameters (a in Y_1, and c in Y_2), which is Fuschian of the third order. Because of its relation to error function we call these solutions spherical error functions.

Initial conditions that uniquely determine Y_1 and Y_2 are the following:

$$Y_1(a=0|c) = c/2\ \exp(-c^2) + (c^2+1/2)\,(\sqrt{\pi/2})\cdot erfc(-c);$$

$$Y_2(c=0|d) = d/2\ \exp(-d^2) + (d^2+1/2)\,(\sqrt{\pi/2})\cdot erfc(-d);$$

From these initial conditions, functions Y_1 and Y_2 are integrated using the following linear differential equation of the third order:

$$\delta_a^3 Y + 2(2a^2-3)\delta_a^2 Y - 2(2a^2c^2 - 2a^4 + 5a^2 - 4)\delta_a Y - 4a^4 = 0,$$

where $\delta_a = ad/da$.

This is a Fuschian equation with only two singularities at $a = 0$ and $a = \infty$, and with a simple asymptotic behavior at $a = \infty$. This

allows for analytic continuation of Y along the real axis from $a = 0$ to the point of evaluation using our methods of fast analytic continuation of solutions of linear differential equations based on recurrences for coefficients of power series expansions, following algorithms and methods from [1]. The corresponding routines were developed on SCRATCHPAD. They boost the performance of integral evaluation by a factor of about 10 compared to Romberg's method.

[The new algorithm for evaluation of these and other similar integrals are even more impressive for scalar execution. In recalculations there is no need to analytically continue a function from the origin, but it can be continued only from the last evaluation place. This significantly decreases the degree of polynomial approximations almost everywhere. On vector and SIMD machines, of course, such varied degree of approximation is more difficult to implement.]

Sometimes physical models have to be adapted to the existing computational environment. While exponents and error functions can be computed in any environment, some machine perform worse than others. GF11 has a hard time at evaluation of exponents and will waste a large number of floating point operations in doing so. This and similar problems in any other vector or parallel machine increase interest in simpler models of solution of the Boltzmann equation than the one tried before. We used SCRATCHPAD to find such explicit models and to develop the corresponding algorithms. We briefly describe one that gave results numerically close to those of rigorous models, in the absence of gravity. Our approach was to maximize the entropy

$$S = -\int p \cdot \ln p \; dx$$

of the distribution function p (subject to a few constraints that fix first moments of p). A well known solution to this problem (in the multidimensional case as well) is given by the Gaussian distribution. We ask for the maximum of entropy over the class of polynomial functions of bounded degree. This leads to a nontrivial system

of transcendental equations on the coefficients that we analyzed in SCRATCHPAD. Such polynomials and their integrals can be used instead of exponents, error functions and spherical error functions. A particularly interesting sequence of such polynomials discovered by us is given by

$$p(x) = A_n(c_n^2 - x^2)^n : -c_n < x < c_n,$$

$$c_n^2 = 2n + 3,$$

$$A_n = (2\sqrt{2n} + 3)^{-2n-1}(2n + 1)!/n!^2.$$

The first two even moments of p are 1, and the entropy is

$$S_n = -log(2c_n) + log((2n + 1)!/n!^2) + 2nlog2 + nG(2n + 2),$$

where $G(z) = \Psi(z + 1/2) - \Psi(z/2)$, and $\Psi(z) = (d/dz)\ \log\Gamma(z)$. S_n approaches, as n grows, the entropy

$$-1/2(1 + \ln(2\pi))$$

of the Maxwell-Boltzmann distribution function

$$\exp(-x^2/2)\ /\ \sqrt{2\pi}$$

3. Code Optimization.

Our codes were prepared for runs on large vector machines or parallel SIMD machines (even more so on MIMD machines).

The 3D code Galaxy consists of several stages. If the number of processors is not too large (namely less than N^2, where N^3 is the grid size), all computations per cell are local. Interprocessor communication is needed only in three parts of the code:

a) to synchronize the time steps from the Courant conditions, i.e. to determine the temperature of the hottest cell (via binary tree or any of its implementations);

b) to transfer fluxes from the boundary of the subgrid occupying a given processor to a boundary of a neighbor subgrid (possibly occupying another processor);

c) in the computation of self gravitation effects, the Poisson equation is solved locally at in each processor separately, but cyclic transposition of the whole grid ($x \rightarrow y \rightarrow z \rightarrow x$) is performed on the whole machine. In complexity this operation is very similar to the transposition of a matrix spread amongst processors.

All communications are negligible in number compared with operation count and with the volume of memory transfers within single processors. Local operations per processor (e.g., in the case when the whole grid occupies a single processor) are completely vectorized with the length of vectorization equal to the total local grid size, and not to the linear dimension, as common in ordinary fluid codes. This gives a very efficient performance on vector machines with long pipes or high latency.

Interesting problems occurred during the development of the code. For vector machines, the performance of a purely vectorized code was never efficient, because of the staggering number of load and store operations occupying valuable cycles. For machines with large register files (such as GF11, which has the total of about 150K words of register file locations and nearly 36 Mbytes of SRAM locations in the machine), breaking the code into a collection of vector macros is inefficient. That is why, using SCRATCHPAD two different versions of the code were developed for the GF11 code generator.

I. Vector code. Here a collection of about 20 vector macros, not unlike the classical SAXPY routines - multiplication of a scalar by a

vector plus another one vector - formed the bulk of the code. Depending on the architecture of the machine this may or may not be an optimal organization (on IBM 3090-VF in view of large memory traffic that cannot be avoided, nearly 30% of all cycles were loads /stores).

II. Vector to Register Code. Code is still totally vectorized, but is built for machines with large register files and aims at the reuse of local variables to reduce the memory traffic.

For such machines as GF11 this is very important, because load or store can occur while floating point operations are performed. Our code thus avoids any hit from memory - to register - to memory traffic, and the code approaches its peak efficiency.

Though versions I and II amount only to polynomial (algebra) manipulation and in principle can be performed by hand, SCRATCHPAD was very helpful for us in the algorithm development. We cannot prove that the total algebraic complexity of our code within a given physical and finite element model is minimal, but it nearly halved the complexity of the code we started from.

Earlier versions of the galactic code, describing the chemistry and thermodynamics, were described by Chiang and Prendergast [2]. Runtime for a 100 x 100 2D code took 10 hours of CPU time on an IBM 3081. Our code for the same problem on IBM 3090 takes about an hour. Unfortunately, to study models and compare with observations, several thousand initial conditions in the 2D case and several hundred in the 3D case have to be run. A slightly bigger dedicated machine is needed and such a machine is GF11.

4. The GF11 Parallel Supercomputer.

The GF11 parallel supercomputer, designed by M. Denneau, was built in Thomas J. Watson Research Center of IBM for purposes of large scale scientific computing. The machine consists of 576 processors organized in a modified SIMD architecture interconnected

through a three stage Benes network. Each processor contains a 256 word Register File, 64 Kbytes of high speed static RAM, 2 Mbytes of dynamic RAM, and is capable of 20 Megaflops. The full machine with over 1 Gbytes of DRAM is capable of a peak 11.52 Gigaflops. The machine has also two special features which circumvent some of the limitations of SIMD. First, each processor can be processing a different variable or array. And second, a set of 8 condition codes can be set as a result of a computation. These codes can then be used to selectively control processors based on results of computation (i.e., for finding the maximum of an array, as above).

The switching network of GF11 can realize any permutation of the 576 processors. It can also realize more general mappings. Switch settings are loaded from a table of 1024 precomputed settings. These settings are computed at compile time and remain constant throughout program execution. Each processor can send and receive a word every 4 cycles.

The code generator package for GF11 underwent several stages of development. At an early stage, to assist in efficient utilization of GF11 for the execution of fully vectorizable code, the code generator was transforming "vector macros" into bursts of microcode. "Vector macros" consisted of vector do-loops of a single subroutine. Long vector loops helped to alleviate inefficiencies due to latencies in the floating/fixed point operations, and interprocessor communication. While it made possible efficient execution of vector codes on GF11 (particularly, in those cases when floating point additions and multiplications were balanced, long vector subroutines can reach nearly 100% of GF11 utilization), it hides an advantage of GF11's large SRAM and Register File. The current code generator can compile efficiently any properly submitted code, suitable for SIMD execution.

Initially the GF11 machine was tested on traditional "benchmarking" codes, to demonstrate its functionality, like multiplication of large matrices spread amongst the processors. Simultaneously with final stages of hardware trouble-shooting, important scientific problems were looked at to choose as first codes to run on GF11 using the

developed code generator. The first project, was the implementation of factorization programs and primality testing. These problems are characterized by high complexity, and, therefore, are used in modern encryption protocols. For example, factoring a 70- digit number can take 10 hours on CRAY. For a 90-digit number, the factoring time jumps to about 10,000 hours of cumulative CPU-time on the network of SUNs. Better parallel algorithms can improve factorization time, and were adopted for GF11 architecture. This part of work was based upon theoretical research [4]. The first stage of the factoring package is now fully operational on GF11, with performance depending only on availability of run time and processors. This number-theoretic work, in addition to its applied interest, is of importance to algebraic geometry because of the use of Abelian varieties and new fast interpolation algorithms. Despite the complicated structure of factorization packages (that include a comprehensive multiprecision arithmetic package developed for GF11), the capabilities of GF11 were utilized at almost 90% without any compromise in the algorithmic structure of the code!

The Galaxy code, embodying all the complexity and instability of a multi-fluid hydrodynamics codes with chemical and thermal interaction between fluids runs on GF11 with an efficiency close to 80%. Its execution on a part of GF11 available to us at this time brings a sustained rate of execution of the Galaxy code to about 5 Gigaflops. One can run simultaneously up to 500 different two-dimensional Galaxy simulations on GF11, or a few three- dimensional problems (with different initial conditions and different laws of star formation and gas cooling).

Contributions To Symbolic Computations.

During the period of the grant much of the work of the Principal Investigators has been devoted to the development of computer algebra programs, packages and their applications in real large scale mathematical and scientific computational tasks. In the PI's work, computer algebra has been used in a multitude of ways beginning with a complete programming environment. One can customize a sophisticated computer algebra system to serve as an interactive interpreter/computer environment for symbolic debugging and program generation. This way, a program of moderate size can be created and debugged starting only from the basic algorithms, customized for a given architecture, and optimized in complexity and performance using symbolic manipulation.

This use of a computer algebra environment was crucial in the development of the first codes to run on the IBM GF11 parallel supercomputer. This machine with 576 20-megaflop processors with a modified SIMD architecture has no compiler in the conventional sense of the word. Performance optimization requires a very specific sequence of floating point operations/communications and individual processors have a very long pipe, (of 25 cycle latency). A specific order of operations can change the performance by many orders of magnitude. In preparing the factorization codes and the multifluid Galaxy codes for GF11, we had to optimize several crucial subroutines – the bignum operation and the fast numerical integration of special functions. The algorithms themselves were derived as a part of our general research in computer algebra implementation, The bignum multiplication is the result of our new complexity studies, and our new algorithms were derived with the help of symbolic computations on high genus algebraic curves [5], [6]. Similarly, new fast algorithms of special function evaluation were initially derived for use in SCRATCHPAD and can be symbolically generated. With these tools available, using SCRATCHPAD as an environment, the whole multifluid galaxy code

was reduced to about 100 lines. (Both for the FORTRAN version and for the GF11 code generator). However, to increase the efficiency of execution on GF11, a large number of automated variations in the algebraic part of the code had to be tested. Without the SCRATCHPAD *environment*, this would have been virtually impossible. As it stands, the efficiency of our code on GF11 (the proportion of cycles used for floating point operations) is between 70 and 80%. [A primary reason for this is the simple excess of additions over multiplications in the algorithms; GF11, as a CRAY and other supercomputers, is 100% efficient if there is multiplication/addition every time.]

The development of new computer algebra algorithms, programs and packages was our primary occupation in the symbolic manipulation field. The particular computer algebra system with the best structure, and potential for supercomputer use is IBM's SCRATCHPAD. We participated in the development of this system, and in the design of its parts, including the graphic interface. Two groups of algorithms were developed within SCRATCHPAD as symbolic techniques, and were used by us in large computational projects.

1. Fast algorithms of integer and polynomial operations.

A basic building block of a computer algebra system is arbitrary length integers and polynomials over integers. Their efficient handling determines the overall performance of the system. The problem of fast integer and polynomial multiplication is closely connected with the general problem of fast convolution over various rings and fields. This class of problems is a well defined area of complexity theory with a history of successful algorithm development and application. For a comprehensive exposition see Winograd [7]. Applications to specific integer and polynomial algorithms are well described in Knuth [8]. Our interest is in purely integer or modular convolution algorithms. We have developed a wide class of fast polynomial multiplication and convolution algorithms based on the interpolation of multivariable

meromorphic functions on algebraic varieties. These new algorithms are of particular practical importance for Riemann surfaces of high genera algebraic curves uniformized by congruence subgroups. Our new fast multiplication and convolution algorithms over finite fields are closely related to algebraicogeometric Goppa codes [9], [5], [6]. Some of our results are summarized below in Appendix 2.

One of the primary complexity results here is the proof of the linear multiplication complexity $\mu_{\mathbf{K}}(n) = O(n)$ of the cost of polynomial multiplication of polynomials of degree n over a finite field $\mathbf{K}$. In many cases we have established tight upper and lower bounds on multiplicative complexities.

For integer and polynomial multiplication algorithms over $\mathbf{Z}$ we have discovered new algorithms improving Schonhage-Strassen multiplicative complexity bounds. For moderate word sizes, specific groups of algorithms were found, lowering the overall complexity (i.e. the total number of primitive operations). These algorithms were applied by us for a range of integers of up to 2 billion decimal places (up to a length of 2^{31} bits). An interesting phenomenon was observed: there seems to be no general algorithm optimal in a large range of word length. It seems, moreover, that the genera of algebraic curves over which there are the best complexity algorithms grows with the word length. In practical applications with an array length of about a million, conventional modular DFT algorithms are quite efficient, but above it new interpolation algorithms are preferable. For applications of our algorithms see further in this Report.

We want to mention that, whenever error free calculations of this nature (or any other image or signal processing) are critical, the use of integer and modular algorithms is the best choice to verify the correctness of the result and the functioning of the computer equipment. A prior statistical error analysis is simply inadequate, and can in practice easily miss serious hardware bugs.

In addition to software development, a hardware specifically designed to perform efficient long integer multiplication and error free integer convolutions was designed and built. We refer to the "Little

Fermat" machine which is a general 256 bit machine [14]. This machine, as its name indicates, was envisioned to be able to do efficient computations in Fermat modular rings:

$$\mathbf{Z}/\mathbf{Z}(2^{2^n}+1).$$

"Little Fermat" was optimized for operations on 256 bit (signed 256 bit) words over $\mathbf{Z}$ or modulo $F_8 = 2^{256}+1$. It also has a flexible programming environment. Long integer convolutions are performed using DFTs over the modular ring $\mathbf{Z}/\mathbf{Z}F_8$ and using recursive convolution algorithms over these DFTs. The construction and debugging of this machine took several years, and started with the help of Dr. Denneau of IBM Research and was conducted by S. Younis of MIT. "Little Fermat" consists of a stand alone unit with six 25' x 25' wirewrapped boards and over 5,000 ICs. "Little Fermat" is fully described in Appendix 4, based on S. Younis' Masters thesis. This hardware project was a one of a kind experience.

2. Differential Equation Solver And Fast Special Function Facility.

The needs of computer algebra systems include arbitrary precision numerical facilities equal to those available to applied scientists on computers in fixed precision. Moreover, the demands of computer algebra systems are much higher, because the user expects a computer algebra system to integrate in closed terms, whenever possible, and to deliver a numerical answer with arbitrary precision in all other cases. For example, the user expects a computer algebra system to be better than any mathematical physics handbooks, such as Abramowitz and Stegun, Bateman Project, Gradsten and Rizick etc. A typical solution is to encode the existing numerical algorithms into a computer algebra system. Often this is impossible in arbitrary precision, for one has to develop algorithms of polynomial complexity to substitute fixed precision recipes.

This computer algebra need was our starting point in 1985. At that time we had to develop algorithms for the computation of algebraic and special functions, and the fast computation of their series expansion. Since then we have developed a variety of algorithms to compute values of solutions of algebraic and differential equations, analytically continued anywhere on the Riemann surface of this solution. We refer to our papers [10], [11], [12], [13], for descriptions of algorithms for fast evaluation and analytic continuation, and examples of applications of these algorithms. The cost of analytic continuation does not differ significantly from the cost of multiplication. For example, we have the following result:

Let $y(x)$ be a solution of a linear differential equation over $\mathbf{C}(x)$, regular at N-bit number $x = x_0$. Given a path γ from x_0 to an N-bit number x_1 (on the Riemann surface of $y(x)$) of length L, one can evaluate $y(x)$ at $x = x_1$ with the full N-bit precision in at most

$$O(M(N) \cdot \{\log^3 N + \log L\})$$

steps (= primitive operations).

Here $M(N)$ is the basic yardstick of the computations: the cost of multiplication of N-bit integers [18]:

$$M(N) = O(N \cdot \log N \cdot \log\log N).$$

[This bound can be significantly reduced – from $\log^3 N$ to $\log N$ – if y(x) has special arithmetic properties, like the E - or G - function condition – satisfied by classical special functions.]

As it turns out, our algorithms are extremely efficient in fixed, machine precision as well. Instead of the slow, inefficient, existing packages and recipes for the computation of nontrivial special functions, the evaluation of numerical quadratures or integrals of differential equations, we use our analytic continuation technique. It gives high precision (accurate for fixed precision) values and simultaneously an approximation polynomial (rational function) valid in a natural domain free of singularities. These schemes, in general, match in their

complexity the hand made polynomial and continued fraction schemes employed to compute elementary transcendental or a couple of well studied special functions (error function etc.). Our schemes can be applied to solutions of arbitrary algebraic or linear differential equations.

Among the applications of our algorithms we want to mention the computation of all monodromy parameters of arbitrary linear differential equations and systems. We are able to solve the direct monodromy problem for linear differential equations with regular and irregular singularities, determining all monodromy and Stokes matrices. This allows us to solve in many interesting cases the inverse monodromy problem - the reconstruction of a linear differential equation by its monodromy data. These facilities led us to interesting applications in several fields of applied mathematics, computational fluid dynamics, computational astrophysics, number theory, complex analysis, Teichmuller spaces, mathematical physics and quantum field theory.

In applied mathematics many applications are related to the efficient numerical construction of conformal mappings of a bounded domain onto a unit circle (Reimann mappings). For this, Schwartz - Christoffel differential equations are used in conjunction with our solution of the accessory parameter problem. We also are solving the uniformization problem for Reimann surfaces using the Poincare - Klein relation between Fuchsian and Kleinian groups and the monodromy of linear differential equations. Here, in addition to interesting number theoretic applications (new irrationality proofs), new results on Teichmuller spaces and accessory (Fricke) parameters were obtained. This includes the disproof of the Whittaker conjecture on accessory parameters and other results, see [13], [15], [11].

A new group of applications of monodromy programs is related to recent numerical and theoretical efforts in quantum field theory. One of the coveted prizes is theoretical and computational physics is the determination of masses of observed elementary particles from the first principles of QCD. This problem is under intensive numerical study

using special purpose parallel supercomputers in several places around the world. The method used here is the Monte Carlo integration for a fixed 3-D plus time lattices. Reliable results are still years away due to the true scale of the problem. New approaches arise from modern string theory, and give one a chance to derive better numerical schemes for the direct computation of functional integrals.

To demonstrate the effectiveness of these schemes one can look at 2 dimensional cases, where better analytical methods are available. In the 2 dimensional case, the string theory models in 1/N-expansion lead to the so-called matrix models (see Mehta [16]). Matrix models allow physicists to count properly the contributions of graphs of arbitrary genera in Feynman integrals. A properly scaled single matrix model leads to a 2 dimensional quantum gravity model, and the multimatrix system describes realistic 3 and 4 dimensional quantum field theories.

The quantum gravity models are related to isomonodromy deformation equations of Painleve type. This discovery was made by physicists in late 1989 (in Princeton and Paris), and led to the expectation that a closed forum solution could be obtained for the scaled limits of partition functions. Unfortunately, physicists could not determine free parameters appearing in the theory or, even, prove the existence of the scaling limit.

The classes of matrix problems here are the following. For the space H_N of N x N Hermitian matrices with the canonical measure $d\mu_N(\phi) = \Pi dRe(\phi_{ij}) \cdot \Pi dIm(\phi_{ij})$, put:

1.
$$Z_N = \int_{H_N} d\mu_N(\phi) e^{-\sum_{k=1}^{\infty} \beta_{2k} \cdot \phi^{2k}/2k}.$$

2.
$$Z_N = \int_{H_N} d\mu_N(\phi) e^{-\sum_{k=1}^{\infty} \alpha_{2k+1} \cdot |\phi|^{2k+1}/(2k+1)}.$$

In particular, in model 2, the problem is the limit of Z_N for the weight $e^{tx+x^3/3}$ when $N = 4\epsilon^{-15}$ and $t = -6\epsilon^{-10} + \epsilon^2 \cdot T$ as $\epsilon \to 0$ (double scaling limit). It is also equivalent to the existence of the limit of the Painleve II equation

$$\lambda_{tt} = 2\lambda^3 + t \cdot \lambda + \alpha$$

for $\alpha = 4\epsilon^{-15}$, $t = -6\epsilon^{-10} + \epsilon^2 \cdot T$, $\lambda = \epsilon^{-5} + \epsilon \cdot \Lambda$ to the Painleve I equation

$$\Lambda_{TT} = 6 \cdot \Lambda^2 + T$$

as $\epsilon \to 0$.

Using our monodromy programs and [17] we were able to determine when the limit exists and the analytic properties of the limit. This is a very interesting area of mathematical and theoretical physics, but its most promising applications are in the numerical realization of QCD computations.

Our programs for the solution of differential equations and special function evaluation were prepared for SCRATCHPAD. SCRATCHPAD is becoming a widely available system and we hope that these programs will be of use. They are not environment dependent and can be used elsewhere.

Large Scale Mathematical Computations.

Overcoming Storage and I/O Bottlenecks.

1. Implementation of basic polynomial time algorithms.

Computers are useful for scientists and engineers only when they perform tasks that people cannot. From this point of view, only large computations matter. Particularly important and difficult are those scientific applications that require large memory and storage and hours, if not days of dedicated supercomputer time. Such computations are a common feature of modern chemistry, theoretical and applied physics, fluid dynamics, X-ray optics and data-processing, as well as in number theory and other areas of theoretical mathematics.

Various classes of seemingly unrelated physical problems share some basic mathematical algorithms and methods of computational solution. Often, an important scientific problem is solvable in polynomial time, but difficulties and various bottlenecks in realization prevent practical solutions. For example, quantum chemistry is one of the most computationally intensive tasks, in which the basic ingredients are familiar polynomial time solvable routines. Often, to tackle a realistic scientific problem one has to work with data of sizes previously considered prohibitive. For example, there is a barrier in modern computers set by 32 - bit addressing, which is already recognized as a major bottleneck in fluid dynamics and other applications, where 4-gigawords of array space is obviously insufficient. Many problems for which efficient algorithms have been constructed, are still left unresolved for lack of storage and speed of existing computers.

We describe here one group of application with a variety of scientific ramifications, where both major obstacles in any supercalculations–namely lack of storage and slow speed–were resolved to our satisfaction even on IBM 3090VF.

The goal of our computation was to apply newly developed (by us)

algorithms of high-precision mathematical computation to the evaluation of classical constants and functions for the purpose of amassing a sufficient database for testing several classes of number theoretic and statistical conjectures. The decimal expansion of π was a natural target because of its historic interest, large existing body of work, and the ability to benchmark the performance of our algorithm against the existing approaches.

The main building blocks of this computation are common to a large variety of other high speed computations. These are the fast algorithms of digital signal processing–fast convolution and fast linear algebra manipulations. From this point of view, the main part of computation was, in a sense, similar to large hydrodynamics, seismological or astrophysical simulations. A comparable size of a grid for physical computation would be about $1000 \times 1000 \times 1000$. The necessary fast convolution algorithms are all descendants of the FFT introduced by Cooley and Tuckey.

In our number-theoretical work there were several additional complications, namely, error free and unbounded dynamic range demands on our convolutional algorithms. Also, since the integrity of the result was the primary concern, major parts of the computations were directed towards result and data validation, verification and correction.

By far the largest part of the computational time in arithmetic with arbitrary precision is spent on bignum multiplication. Other than for relatively short numbers, the high-school method of multiplication should be avoided. A conventional remedy is the use of FFT algorithms to speed up the convolution of digits of factors from which the true digit of the result is reconstructed. It is better to speak of fast convolution algorithms rather than FFT because often the floating point (complex) FFT is less efficient than its modular versions or new fast convolution algorithms.

We look at bilinear form representations of fast multiplication (convolution) algorithms. In all these algorithms one looks at the

product of two bignums written in the radix Rad:

$$A = \sum_{i-0}^{n-1} A_i \cdot \mathrm{Rad}^i,\ B = \sum_{j-0}^{m-1} B_i \cdot \mathrm{Rad}^j,$$

(as polynomials in this radix) as the result of convolution of arrays (A_i) and (B_j):

$$C = A \cdot B = \sum_k \{\sum_{i_j=k} A_i \cdot B_j\} \cdot \mathrm{Rad}^k,$$

(that is, the polynomial product).

The result of convolution $(\vec{A}) * (\vec{B}) = (\vec{C})$, $C_k = \sum_{i+j=k} A_i \cdot B_j$, is computed via the bilinear form algorithm

$$\vec{x} = H_n \cdot \vec{A},\ \vec{y} = H_m \cdot \vec{B}$$

for $H_n \in M_{l\times n}$, $H_m \in M_{l\times m}$; $l \geq n+m-1$, and with the result

$$\vec{c} = G \cdot \vec{z}$$

for $G \in M_{(n+m)\times l}$ and $z_\alpha = x_\alpha \cdot y_\alpha$ for $\alpha = 1,\cdots,l$. Here $l \geq n+m-1$ is the rank of the algorithm, and the whole algorithm can be described as consisting of 3 stages: transforming A and B (by means of linear transformations with matrices H_n and H_m), dot-product of the transformed results $(z_\alpha = x_\alpha \cdot y_\alpha)$, and retransformation (with a matrix G).

In the case of FFT-like algorithms of fast convolution, the matrices H_n, H_m and G are built from primitive roots of unity: $H_n = (w_l^{ij})$, $H_m = (w_l^{ij})$, $G = (w_l^{-ij})$, and the resulting array C is the circular convolution of length l : $C_k = \sum_{i+j\equiv k(l)} A_i \cdot B_j$. The advantage of the FFT-like algorithms is their low complexity compared to that of arbitrary linear transformations of the same size. Usually, l is chosen as a highly composite number, typically a product of powers of a few small primes. Matrices H, G are defined over the ring $\mathcal{G}$ where l is invertible, and where there is a primitive root of unity w_l of order l. The well-known Cooley-Tuckey FFT corresponds to the case $l = 2^r$.

There are usually 3 choices for $\mathcal{G}$ used in practice: 1) the complex number field (with the precision of operation high enough to determine C_k correctly taking into account the loss of $O(\log l)$ digits of precision during the FFT computation–of course, if l is of the order of a billion–i.e. one takes a billion length FFT, up to 60 bits can be lost!); 2) products of finite fields having primitive roots of unity of order l (e.g. for $l = 2^r$ finite fields F_p for primes p are of the special form with $p = s \cdot 2^r + 1$), with integers C_k reconstructed via the Chinese Remainder Theorem; 3) surrogate polynomial or special modular rings like $\mathbf{Z}/\mathbf{Z} \cdot (2^{2^t} + 1)$- -more or less Schonhage-Strassen [18]. In case 3) one is using, strictly speaking, not a pure FFT algorithm but recursive algorithm, where the modular FFT is used to lower the exponent t. A more general approach to fast modular and integer convolution algorithms is described in this Report and in the Appendix 2.

These new algorithms [1], [22] of ours use arbitrary algebraic curves and varieties and represent all fast convolution algorithms as interpolation algorithms on these varieties. Conventional algorithms arise as special cases corresponding to interpolation on a protective line or a circle.

1. Practical implementation of fast convolution algorithms is a source of endless publications. In practice it is a difference in factor from $\log N$ to $\log \log N$ in the speed of execution. For N of the order 10^9 this is a big difference.

An additional complication is a lack of physical memory to support computations except for largest Japanese supercomputers. In our implementations of fast convolution algorithms, all arrays (input, output and scratch) were externally stored on disks or on tapes. 64 megabytes was an upper limit for physical memory used, and we attempted to reduce I/O overhead to a minimum. The difference between the CPU time and I/O overhead was due only to relatively slow channels, and with fast channels was unnoticeable.

2. This approach – to store data externally and bring them directly to the cache, bypassing the physical memory - is old fashioned, and was basically abandoned when supercomputers appeared. It is

clear, however, that supercomputer memory, at least in this country, is too expensive and far too small. This should not stop very large computations. The price of 1 M byte of a hard disk is now about $3 (in small quantities) and standard channels approach 3-5 M bytes/second. Even a couple of such channels is good enough for large jobs. For example, an IBM 3090 with 2 large disks one can solve linear problems of size up to 17K x 17K without noticing any I/O overhead. The runtime of a standard method (LU-decomposition) of solution of such size systems of linear equations is about 8 hours with 100 Megaflop and sustained performance.

Once basic algorithms are available (of multiplication of arbitrarily long numbers, etc.), one can with ease implement a variety of fast number-theoretic and numerical algorithms.

One such algorithms is that of fast evaluation of special functions. We quote one algorithm ("bit-burst") of evaluation for the solutions of linear differential equations:

Theorem 1 *Let $y(x)$ be a solution of a linear differential equation over $\mathbf{C}(x)$, regular at N-bit number $x = x_0$. Given a path γ from x_0 to an N-bit number x_1 of length L, one can evaluate $y(x)$ at $x = x_1$ with the full N-bit precision in at most*

$$O(M(N) \cdot \{\log^3 N + \log L\})$$

bit-operations.

Here $M(N)$ is the cost of multiplication of N-bit integers:

$$M(N) = O(N \cdot \log N \cdot \log\log N).$$

In practice one can apply these algorithms, but can the complexity be improved? In fact, $\log^3 N$ seems to be excessive around $N = 2^{30}$. Complexity can be lowered if arithmetic of the linear differential equation can be invoked. Specifically, it is enough to demand that some branch of $y(x)$ has somewhere a nearly integral Taylor expansion with

algebraic number coefficients. If that is the case, $\log^3 N$ can be replaced by $\log N$. Linear differential equations with such remarkable properties are subject to interesting conjectures. One of them, traced to Siegel, plainly states that $y(x)$ should be reducible to hypergeometric functions.

Generalized hypergeometric functions are defined as power series whose consecutive coefficients satisfy rank one linear recurrence with coefficients being rational functions of indices. In the one-dimensional case, classical notations are those of

$$ {}_mF_n(a_1,\ldots,a_m;b_1,\ldots,b_n;x) = \sum_{N=0}^{\infty} \frac{\Pi_{i=1}^{m}(a_i)_N}{\Pi_{j=1}^{n}(b_j)_N} \cdot \frac{x^N}{N!} $$

functions of parameters $a_i, b_j : (c)_N = (c)\cdots(c+N-1)$. In all arithmetically interesting cases all parameters a_i, b_j are rational numbers.

Constants expressible in terms of values of generalized hypergeometric functions can be called "rank two" constants. The scheme of computation of (truncated) generalized hypergeometric series is based on a simple lower triangular 2×2 matrix recurrence:

$$ \begin{pmatrix} a_n & 0 \\ b_n & c_n \end{pmatrix} = \begin{pmatrix} a_{n-1} & 0 \\ b_{n-1} & c_{n-1} \end{pmatrix} \cdot \begin{pmatrix} A(n) & 0 \\ B(N) & C(n) \end{pmatrix}, $$

for polynomials $A(n)$, $B(n)$, $C(n)$ from $\mathbf{Z}[n]$.

Here c_n is the numerator of the n-th coefficient, b_n is the numerator o thef n-th order truncated (generalized hypergeometric) series, and a_n is the common denominator (of all n terms in the truncated series). Of course, this scheme should not be applied serially, but rather in the form of the tree algorithm, by multiplying adjacent 2×2 matrices–this is equivalent to summing adjacent terms in the truncated series.

Algorithm I. (divide and conquer.) Consider the following scheme of computation of (rational number representation of) truncated generalized hypergeometric series:

$$ \begin{pmatrix} a & 0 \\ b & c \end{pmatrix} = \begin{pmatrix} A(0) & 0 \\ B(0) & C(0) \end{pmatrix} \cdot \begin{pmatrix} A(N-1) & 0 \\ B(N-1) & C(N-1) \end{pmatrix} $$

where $A(\cdot), B(\cdot), C(\cdot) \in \mathbf{Z}[\cdot]$, and $f - \frac{b}{a}$ is the rational number representing N first terms in the generalized hypergeometric series. Then the simplest way to compute a, b and c is the following:

Stage 1, (initialization). Put $M_k = \left(\begin{smallmatrix} A(k) & 0 \\ B(k) & C(k) \end{smallmatrix}\right)$ for $k = 0, \ldots, N-1$.

Stage 2, (multiplication). Put $M_k = M_{2\cdot k} \times M_{2k+1}$ for $k = 0, \ldots, [N/2] - -1$, and $M_k = M_{N-1}$ for $k = (N-1)/2$ for odd N.

Stage 3, (recursion). Put $N =$ ceiling $(N/2)$. If $N > 1$ go to Stage 2, otherwise return $\left(\begin{smallmatrix} a & 0 \\ b & c \end{smallmatrix}\right) = M_1$.

While we recommend this algorithm for many practical implementations, it is not necessarily the best in complexity. More efficient schemes based on these principles were developed for evaluation of solutions of arbitrary linear differential equations by us [12-13], and are discussed above in this report.

The first study of these classes of algorithms and of their extensions needed to compute continued fraction expansions of b/a from the same scheme was done by Gosper in 1972. Since then Gosper has extended his *telescoping technique* [21] to accelerate convergence and to derive new hypergeometric identities.

Bounds on the complexity of Algorithm I and any of its improvements depend considerably on the arithmetic properties of the corresponding generalized hypergeometric series. The worst case bounds are summarized in the following.

Theorem 2 *The cost of computation of $O(N)$ terms of the continued fraction expansion of a number b/a in*

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} = \sqcap_{n=0}^{N-1} \frown \begin{pmatrix} A(n) & B(n) \\ C(n) & D(n) \end{pmatrix}$$

(for polynomial or rational $A(\cdot)$, $B(\cdot)$, $C(\cdot)$, $D(\cdot)$) is at worst

$$O(M(N) \cdot \log^2(N+1)).$$

2. Number - theoretic algorithms.

Let us return to computations of π. Special elliptic curves can be used to construct fast schemes of computations of π, that can be represented as hypergeometric identities. Such elliptic curves possess complex multiplication.

The first major contribution to the theory of period relations since Legendre's time belongs to Ramanujan. According to Ramanujan [23], for any elliptic curve with complex multiplication, there are two linear relations between periods and quasiperiods (between K, K', E, E') with algebraic number coefficients. Substituting these two linear relations into the Legendre identity, Ramanujan arrived at the expression of an algebraic multiple of π as a quadratic function of a single pair of a period and a quasiperiod–of K and E. Moreover, Ramanujan presented this quadratic period relation in terms of a single ${}_3F_2$ function.

We prefer to derive all classes of quadratic period relations from the most general one for the modular invariant $J = J(\tau)$. For this one uses Eisenstein's series.

$$E_k(\tau) = 1 - \frac{2k}{B_k} \cdot \sum_{n=1}^{\infty} \sigma_{k-1}(n) \cdot q^n$$

for $\sigma_{k-1}(n) = \sum_{d|n} d^{k-1}$, and $q = e^{2\pi i \tau}$. The standard theory of complex multiplication states that for an arbitrary elliptic curve over $\overline{\mathbf{Q}}$ with complex multiplication by $\sqrt{-d}$, and with periods $\omega_1, \omega_2 : \tau = \omega_1/\omega_2 \in H$, all ratios $E_{2n}(\tau) : (\omega_2/2\pi i)^{2n}$ for $n > 1$ are algebraic numbers. Ramanujan proved a new algebraicity statement for a non-holomorphic (Kronecker's) version of $E_2(\tau)$:

Lemma 1 *If $\tau \in \mathbf{Q}(\sqrt{-d})$, then $\Gamma(1)$-invariant non- holomorphic series*

$$s_2(\tau) \stackrel{\text{def}}{=} \frac{E_4(\tau)}{E_6(\tau)} \cdot (E_2(\tau) - \frac{3}{\pi \mathrm{im}(\tau)}),$$

has an algebraic value from a Hilbert class field $\mathbf{Q}(\sqrt{-d}, J(\tau))$.

If an elliptic curve E has a complex multiplication by $\sqrt{-d}$, and periods and quasi-periods $\omega_1, \omega_2, \eta_1, \eta_2$, where $\tau = \omega_1/\omega_2 = (1 +$

$\sqrt{d})/2, d > 0, d \equiv 3(4)$, and if g_2, g_3 are Weierstrass invariants of E that are algebraic integers from $\mathbf{Q}(\sqrt{-d}, J(\tau))$, we arrive at Ramanujan's quadratic period relations from Legendre identity ($\omega_2\eta_1 - \omega_1\eta_2 = 2\pi i$):

$$\omega_2\eta_2\sqrt{-d} + \omega_2^2(\sqrt{-d}\frac{3g_3}{2g_2}s_2(\tau)) = 2\pi i.$$

Next one applies a special case of the Clausen identity:

$${}_2F_1(1/12, 5/12; 1; z)^2 = {}_3F_2(1/6, 5/6, 1/2; 1, 1; z).$$

This equation representing quadratic period relations of elliptic curves with complex multiplication in ${}_3F_2$-form is [25]:

$$\sum_{n=0}^{\infty}\{\frac{1}{6}(1-s_2(\tau))+n\}\cdot\frac{(6n)!}{(3n)!n!^3}\cdot\frac{1}{J(\tau)^n} = \frac{(-J(\tau))^{1/2}}{\pi}\cdot\frac{1}{d(1728 - J(\tau))^{-1/2}}.$$

Here $\tau = (1+\sqrt{-d})/2$. The largest one class discriminant $-d = -163$ gives the most rapidly convergent series among those series where all numbers on the left side are *rational*:

$$\sum_{n=0}^{\infty}\{c_1 + n\}\cdot\frac{(6n)!}{(3n)!n!^3}\frac{(-1)^n}{(640,320)^{3n}} = \frac{(640,320)^{3/2}}{163\cdot 8\cdot 27\cdot 7\cdot 11\cdot 19\cdot 127}\cdot\frac{1}{\pi}. \tag{1}$$

Here

$$c_1 = \frac{13,591,409}{163\cdot 2\cdot 9\cdot 7\cdot 11\cdot 19\cdot 127}$$

and $J(\frac{1+\sqrt{-163}}{2}) = -(640,320)^3$. This is one of a few formulas that Ramanujan missed.

Ramanujan provides instead of this a variety of other formulas connected mainly with the three other triangle groups commensurable with $\Gamma(1)$. These are

$${}_3F_2(1/2, 1/6, 5/6; 1, 1; x) = \sum_{n=0}^{\infty}\frac{(6n)!}{(3n)!n!^3}(\frac{x}{12^3})^n$$

$${}_3F_2(1/4, 3/4, 1/2; 1, 1; x) = \sum_{n=0}^{\infty}\frac{(4n)!}{n!^4}(\frac{x}{4^4})^n$$

$$ {}_3F_2(1/2,1/2,1/2;1,1;x) = \sum_{n=0}^{\infty} \frac{(2n)!^3}{n!^6}(\frac{x}{2^6})^n $$

$$ {}_3F_2(1/2,2/3,1/2;1,1;x) = \sum_{n=0}^{\infty} \frac{(3n)!}{n!^3} \cdot \frac{(2n)!}{n!^2}(\frac{x}{3^3 \cdot 2^2})^n. $$

Representations similar to (1) can be derived for any of these series for any singular moduli $\tau \in \mathbf{Q}(\sqrt{-d})$ and for any class number $h(-d)$, thus extending Ramanujan's list ad infinitum. Ramanujan's own favorite on this list,

$$ \frac{9801}{2\sqrt{2\pi}} = \sum_{n=0}^{\infty} \{1103 + 26390n\} \frac{(4n)!}{n!^4 \cdot (4 \cdot 99)^{4n}}, \tag{2} $$

was used by Gosper in 1985 for his record computation of over 17 million terms in the continued fraction (and decimal) expansion of π.

The complexity

$$ O(M(n) \cdot \log^2 n) $$

of computation of $O(n)$ leading digits (or $O(n)$ terms in the continued fraction expansion) of a rank 2 arithmetic constant is the "worst case scenario".

This is typical for numbers, whose convergent series approximation has growing factorials according to the rank two (telescopic) representation of a number, but convergence of the series is only geometric. This was typical for early (pre 1973) formulas of computation of π, such as

$$ \pi = 16 \cdot \tan^{-1} \frac{1}{5} - 4 \cdot \tan^{-1} \frac{1}{239}, $$

(of John Machin 1706), which was used until 1981, with the series approximation of $\tan^{-1} x$:

$$ \tan^{-1} x = \sum_{n=0}^{\infty} \frac{(-1)^n x}{(2n+1)x^{2(n+1)}}, $$

– another hypergeometric function.

Similarly, one can compute in a scheme of rank two, values of (approximations to) hypergeometric or special functions with complexity $O(M(n) \cdot \log^2 n)$. However, values of E- functions, such as e, yield $O(n \cdot \log n)$ terms in the decimal expansion within this scheme, while numbers like π get only $O(n)$ leading digits.

The first published algorithm that allow for computation of π's $O(n)$ digit with complexity $O(M(n) \cdot \log n)$ belongs to Salamin (1972). It was also derived independently by R. Brent.

The Brent-Salamin method [24] is still superior from the point of view of storage requirements to other similar recently developed algorithms (derived from modular equations by Borweins).

All these schemes have to be run with a full precision (even plus extra $O(\log n)$ digits) to achieve $O(n)$ digits in π.

Unlike such "floating-point" schemes (but, in these schemes, contrary to Newton's there is no "error-correction" on the next iteration), the telescoping algorithms of rank 2 (or higher) allow us to add more terms, or digits, to the expansion at any moment.

Salamin-Borwein schemes were used by the Japanese mathematicians Kanada and his collaborators, and by D. Bailey to establish record π calculations (Kanada-Tamura 16 MD (1983), D. Bailey 29.36 MD (1986), Kanada 134 MD (1987), Kanada 201 MD (1988) [26]).

However in 1985 the record belonged to Gosper, who used only a Symbolics workstation, and a telescoping algorithm of rank 2.

We have calculated by May 1989 480 MD digits of π using identity (1) for a multiple of π and a "telescoping" rank 2 method, with congruences to supplement computations with verifications.

Code was prepared for 3 mchines:

1) GF 11;

2) IBM 3090-VF;

3) CRAY 2.

The 480 MD computation was completed on IBM 3909 and on CRAY 2 in MSC.

By the end of July, 1989 we had computed over 1,011,000,000 digits of π on IBM - 3090 VF [27]. We *stopped* our calculations in

September with over 1,130,000,000 decimal digits of π.

These last computations were performed at the T.J. Watson IBM Research Center, Yorktown Heights, on IBM - 3090/200 and IBM - 3090/600.

There are some applications of explicit computation of π. E.g., we used it to prove that for all rational p/q with integral p, q we have

$$|\frac{\pi}{\sqrt{3}} - \frac{p}{q}| > |q|^{-5.793}$$

whenever $|q| > 1$. For this approximation to $\pi/\sqrt{3}$ were checked below 10^{10^8}.

3. Computational environment. Performance and error protection.

Computational requirements for π and other similar supercalculations:

a) a large work area accessible during the computation; in our case one Gigabyte of data was input/output with 4 Gigabytes of scratch space. The total work area was between 4 to 6 Gigabytes (with 4 Gigabytes as a lower bound in the algorithm);

b) fast access to blocks of the full work area. The ratio of paged storage to the number of computations performed per element was relatively low–on the order of $\log N$.

As a consequence of a) and b) the problem was clearly I/O bound. An obvious solution, usually implemented for smaller computations, is to run the whole problem in the physical (primary) memory. For problems of our size it is unfeasible on any existing machine, because even though the local workspace is slightly over 4 Gigabytes, all arrays required to complete the job (including the final output) take almost 10 Gigabytes.

(These are rational approximations to π before conversion to floating point decimal expansion and a couple of Gigadecimal constants: rational approximation to $1/\sqrt{640320}$, and quotient and remainder in the integer division).

Similar I/O restrictions arise when one solves linear algebra or grid problems of large size. Typical targets here are:

$$32K \times 32K$$

dense linear problems or

$$1K \times 1K \times 1K$$

grids. In some of these cases there are more favorable I/O requirements (for example, in matrix multiplication or solution of linear problems one expects a higher ratio of operations versus size of paged blocks).

With the help of Gordon Slishman (IBM Research) several I/O protocols were tested and a couple of them were used: one under MVS and two under CMS and CMS/XA. Common features of usage of these I/O are:

i) to treat disk storage as "scratch memory", accessing blocks (arrays) suitable for efficient vector CPU operations;

ii) to have all I/O operations FORTRAN callable with minimal use of assembler only inside I/O calls themselves;

iii) to place high emphasis on error detection/verification;

iv) to use the ability of parallel (programmable) I/O access to improve I/O performance.

This approach has an additional advantage, for it allows "people's parallelization": when several jobs access the same storage it allows for several user ID to work on different parts of the problem, and in the case of MVS even to synchronize their operations.

The integrated environment of the 3090 mainframe provided an ideal proving ground for testing a variety of options. Different operating systems coexist and there are several efficient ways of using them on the same problem.

For example, tape turned out to be a very economical and efficient storage medium. The IBM 3480 tape system supports in realistic conditions 2.5 MB/second transfer rate inside the application and

holds over 200 MB per cartridge. If not for operators inconvenience, the whole job coud be run with 9 tape cartridges attached to the machine and changed during the course of computations (a total 30 cartridges/session for several sessions).

Instead tapes were used only for back-up and data exchange between operating systems. The primary storage was a 3380 disk system.

The peak performance of Slishman I/O routines on 3380 under MVS was about 2.6 MB/sec. Here the basic data unit was the whole track (or a cylinder = 15 tracks = 712KB). Arrays were paged in and out of main memory in installment varying between 32K and 64K words, delivering an average I/O performance under MVS of about 2MB/sec. On CMS different routines were used that did not bypass a huge system overhead. On a congested day under CMS a single disk was delivering 0.5 MB/sec. Still, several independent disks and proper partitioning of data into "banks" – all by itself an interesting integer programming problem – delivered sufficient performance.

The "true" time of CMS computations of 1 Gigadecimal expansion of π was about 120 CPU hours. This count does not include the I/O count (because the system barely registered proper timing); the time for tape operations; failed sessions, aborted or discontinued jobs; jobs with possible I/O problems; and time for the routine (algorithm) development. As a time frame: to compute 480 million digits took 5 months of development and computation; to complete over 1 Gigadecimal took 2 more months.

During these computations we were treated as any other batch job user (and one knows what this means).

The ability to perform scientific computations on many Gigabytes of data is very important for modern supercomputers. To give one a feeling for an average problem, let us look at one example computed using Slishman I/O.

A system of 17,000 of (full) linear equations (with dense 17,000 $\times$ 17,000 matrix and 24 different right hand sides) was solved. IBM 3090S took 8.7 CPU hours and 8.9 hours of elapsed time on a single

processor, delivering about 104 Megaflop sustained. (The data of this problem resided on 2 disks.)

In the course of a large scale computation, one can encounter the following types of errors:

I. Hard(ware) bugs–in a fully simulated and debugged hardware, one encounters only very rare (repeatable) patterns that can occur as a result of specific combination of instructions.

II. Soft(ware) bugs–in addition to programmer's errors, these are compiler errors, usually occurring in new implementations of library calls.

III. Hard and soft memory errors–relatively rare occurrences of a 2-bit error in a large memory system, guaranteed to correct all single-bit errors. When detected, such errors interrupt any running computations. A truly horrible combination is a triple bit error, or a peculiar physical damage to the memory core.

IV. More typical is an error in I/O, or in a massive storage device. Large disks crashes and corrupted tapes are not just a nightmare but a reality, that *often occur in a multi-month computation.*

Protections can vary: one can run modular verification of all files (more than just a checksum computation); if you can store everything with extra redundancy, in excess of that provided by your hardware. While local operation can be quite well checked this way, if one knows the answer in advance (or at least can check it with an extremely high degree of probability), one will be safer to run multiple modular checks. If a source of a mistake is a specific repeatable pattern, chances are that a simple checksum check may fail to detect it.

For local operations we used several sets of primes to verify all bignums operations modulo these primes, to bring any local error to a probability below 10^{-290}.

4. Statistical analysis of number-theoretic expansions.

Why should one bother to compute π or run any similar super-calculations? An immediate purpose is the hardware and software testing–a crucial part of verification of a complex mixture of devices and problems of super computer machines.

Also, we must emphasize the need of such calculations for the benefit of number theory. Fixed-radix representations of classical constants is largely an untracked field of diophantine approximations, and it might be open to a better theoretical understanding with more identities and numerical work. One can even argue that digits problems such a normality are easier to analyze than continued fraction problems, because they are "additive" in nature, and arise from specialization of power series representation of functions.

Do we have enough information to start making some definitive statements? We would like some more. Let us look at statistics.

A real number $x = \epsilon_1\epsilon_2\epsilon_3\ldots = \sum_{n=1}^{\infty} \epsilon | n \cdot R^{-n}$ in the base (radix) $R : \epsilon_i = 0,\ldots,R-1$; is called normal in base R if for any $l \geq 1$ and any fixed sequence $\Delta = \delta_1\ldots\delta_l$ of $0,\ldots,R-1$ of length l we have: $\lim_{n\to\infty} \frac{1}{n} N_n(\Delta) = R^{-1}$, where $N_n(\Delta)$ denotes the number of the indices i, $l \leq i \leq n$, for which: $\epsilon_i\ldots\epsilon_{i+l-1} = \delta_1\ldots\delta_l$. Borel (1907) proved that almost all numbers are normal in all bases.

According to a folklore conjecture all irrational classical constants $(\sqrt{2}, e, \pi, e^{\pi}\ldots)$ are normal. Overwhelming numerical evidence supports this conjecture though no proof is known.

Still normality is not "true" randomness, and, following Kolmogorov - Chaitin, no classical constant has a random sequence of its digits (due to its relatively low complexity). What are statistical rules that allow one to distinguish truly random sequences from expansion of the classical constants? We think that one of such rules is that of the iterated logarithm (Khintchine) and its generalization.

In this law, due to Khintchine, for almost all real numbers x,

$$\limsup_{n\to\infty} \frac{\sum_{i=1}^{n} \frac{R-1}{2} - \epsilon_i(x)}{\sqrt{\frac{R^2-1}{12}} \cdot \sqrt{2n \log\log n}} = +1;$$

and similarly for lim inf, with +1 replaced by -1. This law fails on known artificially constructed normal numbers. E.g., it fails on the best known normal number - Champerowne numbers (in which all integers written down one after another), $c = .123456789101112\cdot(R = 10)$.

The law of the iterated logarithm and its specializations suggest to look at a random walk (Brownian motion) generated by sequences of digits – an idea that was proposed as early as 1965 by R. Stoneham. Such a formulation allows one to construct many fractal objects (landscapes, etc.) from "random" sequences, such as digit expansions.

We report some statistical data on a billion decimal digit expansion of π below in **Appendix 3**, together with details of hypergeometric identities and congruences needed in computations of classical constants.

Statistical observations of π (and some algebraic numbers) reveal the following:

First, the decimal expansion of π in billion plus range passes with flying colors all classic randomness tests: frequency, χ^2, pocker, arctan law, ... etc.

Secondly, the iterated logarithm test leaves yet unresolved the "generic" nature of π–we are not yet certain that visible deviations from the law of iterated logarithms are statistically impressive. On the other hand, we tested the iterated logarithm law on conventional pseudorandom number generators: they all fail drastically (but in different ways) well below their periods.

Thus, tentatively speaking, π expansion "looks more random", than anything else man-made, but, perhaps, "not random enough". Practical importance of the use of Algorithms I in random number generation and inscription protocols remain to be seen.

There is even a slight disagreement about the validity of the Chaitin - Kolmogorov test for π. The basic formula for π is simple, but the total complexity is a different issue: for example, Kolmogorov called a set of N elements of low complexity, if it can be generated with a program of total length $\log N$. While the length of a π code is a couple of hundred lines, for its execution the full size $O(N)$ storage has to be available–we do not know how to generate a single (N-th) digit of π without keeping somewhere $O(N)$ storage of the previous digits, (though our algorithm allows us to add $N+1$ - st digit to previous N.)

While it takes a delicate statistical analysis to distinguish a particular digit expansion from a generic one, it seems that looking at their continued fraction expansions, classical constants can be easier to distinguish from generic numbers. For example, this is true at least for those classical constants, for which explicit continued fraction expansions are known. (Note that according to our conjecture, all such constant are reduced to hypergeometric constants.)

This is the case of e and e^2 and $\sqrt{2}$. For other classical constants, even very similar to these, no explicit continued fraction expansion is expected. For example, this is the case of e^3 or e^4. We can ask the following question: What kind of continued fraction expansion of an irrational number is "usual" and what kind is "unusual"?

We studied applications of the law of the iterated logarithm.

Let $x = [0; a_1, a_2, \ldots]$ for $a_i = a_i(x)$ and x in $(0,1)$. Then for almost all x we have the Khintchine theorem

$$\frac{1}{N}\sum_{i=1}^{N} \log a_i \longrightarrow \log K. \qquad \text{(KK)}$$

This "Khintchine" test is often too crude to distinguish a "mildly unusual" number x from a "very unusual" number x. That is why we suggest looking at a better form of Khintchine's law. Namely, according to the law of the iterated logarithm (or, rather, a version of central limit theorem), for almost all numbers x in $(0,1)$ we have:

$$\limsup_{N\to+\infty} \frac{\sum_{i=1}^{N} \log\{a_i/K\}}{\sqrt{2N \log\log(N)}} = K^{(2)}, \qquad \text{(KK)}$$

(and, similarly,

$$\liminf_{N\to+\infty} \frac{\sum_{i=1}^{N} \log\{a_i/K\}}{\sqrt{2N \log\log N}} = K^{(2)}),$$

for a constant $K^{(2)} > 0$. The only expression of $K^{(2)}$ we know is such

$$K^{(2)} \stackrel{\text{def}}{=} \lim_{N\to\infty} \frac{1}{N} \cdot \int_0^1 \Big(\sum_{i=1}^{N} \log\{\frac{a_i(x)}{K}\}\Big)^2 \frac{dx}{\log 2(1+x)}.$$

The expression (KK) of Khintchine's law is a much better test than the old (K) test. This is particularly clear for such "explicit" constants as e^2 or $\sqrt{2}$.

We looked at several classes of classical constants, such as:

e^3, e^4, $\pi/\sqrt{2}$, $\pi/\sqrt{640320}$, $\sqrt[3]{2}$ or other Brillhart cubic numbers.

Interestingly enough, the law of the iterated logarithm as applied to continued fractions reveals statistical irregularities for many classical constants. To get the continued fraction expansion data one should start with a decimal (or binary) expansion. The cost of continued fraction computation is of the same order as the cost of the original computation of π.

We suggest after a preliminary analysis of the data, that classical (irrational) constants do not obey the modified Khintchine test. In fact, a challenging problem here is to find a classical constant whose continued fraction expansion looks generic (while for decimal expansions normality is expected).

Conjectures of Siegel (see [27]) describe the class of arithmetical *constants* defined by linear differential equations in terms of hypergeometric constants. This class of constants we refer to as "rank two constants". While this class of constants (or functions) contains many important arithmetical and geometric objects, not everything of number-theoretic significance can be described in terms of linear differential equations directly. In particular, one cannot claim that all "explicit" continued fraction expansions of "explicit functions" are reducible to the ordinary hypergeometric. For example, another group of explicit continued fraction expansions arises from q-basic extensions of hypergeometric functions. (Among explicit continued fraction expansions arising in this context are Rogers-Ramanujan expansions, and other q-basic fractions related to q-basic generalizations of McDonald's identity.)

Recently, we found a new group of explicit continued fractions that can be described as elliptic generalizations of both Gauss and q-basic continued fractions. They are "explicit" in a sense that all element of the continued fraction expansion are explicit functions of the index–

theta functions of the index. In the degenerate case (when θ-functions are reduced to trigonometric functions), our elliptic generalizations are reducible to q-basic continued fractions. (In the degeneration $k^2 \to 0$ these continued fractions are reducible to Gauss's hypergeometric continued fraction for ${}_2F_1$ functions.)

Ordinary Gauss continued fractions have the form

$$\frac{{}_2F_1(a,b;c;z)}{{}_2F_1(a,b+1;c+1;z)} = 1 + \frac{a_1 z}{1+}\frac{a_2 z}{1+}\ldots\frac{a_n z}{1+}\ldots$$

with

$$a_{2n+1} = -\frac{(a+n)(c-b+n)}{(c+2n)(c+2n+1)},$$

$$a_{2n} = -\frac{(b+n)(c-a+n)}{(c+2n-1)(c+2n)}.$$

(This continued fraction and its specializations give very fast computational scheme for many special functions.)

The next level of continued fraction identities is given by Stieljes-Rogers expansions

$$\int_0^\infty cn(u,k)e^{-zu}du = \frac{1}{z+}\frac{1^2}{z+}\frac{2^2k^2}{z+}\frac{3^2}{z+}\frac{4^2k^2}{z+}\ldots.$$

Our new elliptic generalizations of hypergeometric continued fractions depend on five parameters. One of the simplest specializations gives the following explicit continued fraction in elliptic functions of the indices.

$$\int_0^\infty \wp'(t+z)e^{-zz}dz = \cfrac{c_0}{x+B_0-\cfrac{C_1}{x+B_1-\cfrac{C_2}{x+B_2-}}}$$

Here

$$C_n = (n+1)^2(\wp(t) - \wp((n+1)t));$$

$$B_n = (n+1)\varsigma(t) - \varsigma((n+1)t) + (n+2)(\varsigma(t) + \varsigma((n+1)t) - \varsigma((n+2)t)).$$

References.

[1] D.V. Chudnovsky, G.V. Chudnovsky, On expansion of Algebraic Functions in Power and Puiseux Series I and II, J. of Complexity, v. 2 (1986), 271-294; v. 3 (1987), 1-25.

[2] W.-H. Chiang, K.H. Prendergast, Numerical Study of a Two-Fluid Hydrodynamic Model of the Interstellar Medium and Population I Stars, Astrophys. J., v. 297 (1985), 507-530.

[3] J. Beetem, M. Denneau, D. Weingarten, The GF11 parallel computer, in Experimental Parallel Computing Architectures, Elsevier, 1987.

[4]. D.V. Chudnovsky, G.V. Chudnovsky, Sequences of Numbers Generated by Addition in Formal Groups and New Primality and Factorization Tests, Adv. Appl. Math. v.7 (1986), 187 - 237.

[5]. D. V. Chudnovsky and G. V. Chudnovsky, Algebraic complexities and algebraic curves over finite fields, *Proc. Natl. Acad. Sci. USA* **84**, (1987), 1739-1743.

[6]. D. V. Chudnovsky and G. V. Chudnovsky, Algebraic complexities and algebraic curves over finite fields, *J. Complexity*, **4**, (1988), 285-316.

[7]. S. Winograd, Arithmetic Complexity of Computations, CBMS - NSF Regional Conf. Series Appl. MAth., SIAM Publications v. 33, 1980.

[8]. D. Knuth, "The Art of Computer Programming", v. 2 Addison-Wesley, Reading, , 1981.

[9]. V.D. Goppa, Codes and information, Russian Math. Survey, 39 (1984), 87-141.

[10]. D.V. Chudnovsky, G.V. Chudnovsky, On expansion of algebraic functions in power and Puiseux series, Part I and II, J. Complexity, 2(1986) 271-294; and 3(1987), 1-25.

[11]. D.V. Chudnovsky, G.V. Chudnovsky, Computer assisted number theory, *Lecture Notes Math., Springer, New York*, **1240**, (1987), 1-68.

[12]. D. V. Chudnovsky and G. V. Chudnovsky, Computer Algebra in the Service of Mathematical Physics and Number Theory, in "Computers and Mathematics, Stanford 1986", M. Dekker, New York, 1990.

[13]. D. V. Chudnovsky and G. V. Chudnovsky, Approximations and complex multiplication according to Ramanujan, in "Ramanujan Revisited", Academic Press, New York, 1988, 375-472.

[14]. D. V. Chudnovsky, G. V. Chudnovsky, M. M. Denneau, S. G. Younis, A design of general purpose number-theoretic computer, in Proceedins of Supercomputing '88, v. 2, 498-499.

[15]. D. V. Chudnovsky and G. V. Chudnovsky, Transcendental methods and theta-functions, in "Proc. Symp. Pure Math.", American Mathematical Society, Rhode Island, 1989, (in press).

[16]. M. L. Mehta, Random matrices, Academic Press, 1977.

[17]. D. V. Chudnovsky, G. V. Chudnovsky, Laws of composition of Backlund transformations and the universal form of completely integrable systems in dimensions two and three, Proc. Natl. Acad. Sci. USA, 80, 1983, 1774-1777.

[18]. A. Schonhage and V. Strassen, Shnelle Multiplikation Grosser Zahlen, *Computing*, 7, (1971), 281-292.

[19]. D. V. Chudnovsky, G. V. Chudnovsky, M. M. Denneau, K. Prendergast, Supercalculations on GF11: a case study of galaxy code, in Procceedings of Supercomputing '89, v. 2, 317-326.

[20]. D. V. Chudnovsky, G. V. Chudnovsky, M. M. Denneau, Regular graphs with small diameter as models for interconnection networks, in Supercomputing '88, v. 3, 232-239.

[21]. W. Gosper, Continued Fraction Arithmetic, *preprint MIT AI Lab.*, (1972).

[22]. D. V. Chudnovsky and G. V. Chudnovsky, Elliptic formal groups over $\mathbf{Z}$, and $\mathbf{F}_p$ in applications to number theory, computer science and topology, *Lecture Notes Math., Springer, New York*, **1326**, (1988), 11-54.

[23]. S. Ramanujan, Modular equations and approximations to π, *Quart. J. Math.*, **45**, (1914), 350-372.

[24]. E. Salamin, Computation of π using arithmetic- geometric mean, *Math. Comput.*, **30**, (1976), 565-570.

[25]. D. V. Chudnovsky and G. V. Chudnovsky, Use of computer algebra for diophantine and differential equations, in "Computer Algebra", M. Dekker, New York, 1988, 1-82.

[26]. Y. Kanada, Vectorization of multiple-precision arithmetic program and 201,326,000 decimal digits of PI calcilation, in Supercomputing 88, v. 2, 117-128.

[27]. D. V. Chudnovsky and G. V. Chudnovsky, The computation of classical constants, *Proc. Natl. Acad. Sci. USA* **86**, (1989), 8178-8182.

IV. Parallel Algorithms for the Finite Element Method.

Existing finite element methods, algorithms and application programs have been developed in the milieu of the serial or single processor computer. Substantial efforts are currently being made to modify their computational aspects in order to take advantage of the power inherent in the newer generations of parallel multiprocessor computers. The parallelizing techniques being explored tend to be derived from or are closely related to serial concepts. Parallelism is difficult to achieve from this direction. The present work develops and implements a new class of parallel algorithms for the finite element method based upon the use of projections on finite-dimensional spaces. Projections provide a formal and general mechanism for decomposing or partitioning a problem into a collection of smaller related problems which can be treated in parallel.

The decomposition is based upon the ideas formulated by the authors in ([1]-[4]) of recasting the finite element method into a collection of mathematical projections of a given problem with each projection defining an autonomous subproblem related in a well-defined manner to other subproblems. Each subproblem is treated as a process and consists of independent computational instructions and required data dependent upon other processes. The data dependence between processes defines an interprocess communication network. Processes and their communication network are the basic conceptual elements in the parallel implementation. Once these elements have been defined, physical processors (CPU's) can be assigned to execute processes and to implement the network.

1. Performance analysis.

Our algorithms can be programmed in a wholly asynchronous mode and have been implemented and studied on a shared memory multiprocessor computer with up to 24 processors. On large test

problems they have been shown to produce almost perfect speedup and to be extremely efficient.

There are a number of ways to measure the performance of algorithms designed to operate in a multiprocessor environment. One of the most intuitive and commonly used measures is the so-called speedup:

$$(speedup)S_N = \frac{T_1(\ execution\ time\ using\ one\ processor)}{T_N(execution\ time\ using\ N\ processors)}.$$

Perfect speedup, $S_N = N$, is obtained when a program runs N times as fast when N processors are used as when only one processor is used. Speedup is reduced by data dependencies, by communication and/or by synchronization overhead. Experimental results based upon the algorithms were developed on a Sequent Corporation 24 Processor Balance 21000 computer at the Argonne National Laboratory. The timings were obtained by solving Poisson's equation in a rectangular domain. A simple rectangular element was adopted for this problem. During execution, the main program reads the input which includes the required data and the number of processors to be employed. After the completion of the input, the main program distributes memory for variables and sets up the parallel system among the employed processors. Once the solution converges, the main program stops the parallel system and writes the output file.

To test the effect of parallelism on performance, calculations were conducted with varying numbers of processors and different size problems. Computed results were obtained from iterations which continued until the maximum absolute value of the projective residual component was less than a prescribed tolerence. The most interesting aspect of the performance is the speedup shown in Figs. (2) - (4) comparing the theoretical (dashed line) and experimental (solid line) speedups. Fig. (2) indicates the speedup obtained by solving a 60 x 60 grid with 57 partitions by the residual propagation scheme, in which it can be seen that the parallel system provides excellent speedup on different numbers of processors except on the example done by 23 pro-

cessors for which the scaled ratio of 2.48 provides less than 90 percent efficiency as seen from Fig. (1). This lower speedup can be improved by raising the scaled ratio. Fig. (1) is a typical curve, in the case of unit communication width, of efficiency vs. scaled ratio; it was generated from experiments on the test problems. In all cases it was found that once the scaled ratio exceeded about four, the efficiency approached 100 percent. The results in Fig. (3) were obtained by the mixed scheme for the same problem and it too shows excellent speedup. Fig. (4) presents results obtained by solving a 100 x 100 grid with 75 partitions by the residual propagation scheme. In this example, the minimum scaled ratio is greater than 3, and the measured speedup is very close to the theoretical one. These results show that the parallel algorithms developed are highly efficient.

Parallel algorithms for finite element analysis in our approach are based upon the concept of successive projection approximations. The successive projection approach appears to be a powerful tool for achieving parallel efficiency. The algorithms provide almost perfect speedup within the range of processors (24) used in this study. The method enables large problems to be decomposed into a number of smaller problems which can be treated in parallel and can serve a prototype for future development. The present work has developed two computational schemes: the residual propagation scheme and the mixed scheme. Both provide efficient parallel computations. The mixed scheme accelerates the convergence rate but also raises the computational complexity. In a system with a highly efficient method for accessing the data in sparse matrices, the residual propagation technique is faster than the mixed scheme. During the solution procedure, both schemes perform asynchronously so that no processors are idle. A further enhancement has been the development of a data storage method for a general sparse matrix which saves CPU time and memory space. The parallel algorithms developed in this work are clearly powerful and useful, but there still remain significant modifications that could make them even more versatile, i.e., alternatives using overlapped projections or by modest changes for implementa-

tion on message-passing machines. Computational experiments are being conducted to assess the impact of such features on speedup and storage.

2. Performance on an IBM 3090-300E VE.

Timing results were obtained on an IBM 3090-300E with 3 processors by using our parallel algorithms in ([1] - [7]). The test problems were defined by a partial differential equation:

$$A\frac{\partial^2\phi}{\partial x^2} + B\frac{\partial^2\phi}{\partial y^2} + 2 = 0,$$

in two different domains subject to the boundary condition $\phi = 0$. The performance details are shown in Figures 5 and 6.

The results in Fig. 5 were obtained by meshing the domain into 10000 linear rectangular elements and using the control parameters: partitions = 12 and maximal relative residual ≤ 0.000125. The results in Fig. 6 were obtained by meshing the domain into 8200 linear rectangular elements and using the control parameters: partitions = 20 and maximal relative residual ≤ 0.00125. Both results produced almost perfect speedup. However, a system of more partitions requires more iterations to complete the solution as discussed in ([1] and [2]). The idle time in each task in both tables was according to wall-clock time so that the idle times recorded were dependent on the system loading. Thus the speedups indicated are conservative–they would be even better if the multiprocessors were dedicated to a single job execution. The results in Fig. 5 illustrate a rapid convergence, while the second case was an example of a slow rate of convergence requiring more iterations. Thus the speedup is independent of the rate of convergence.

The object of this research was to demonstrate the outstanding performance of an IBM 3090 multiprocessor computer in executing the parallel algorithms for the finite element method. The measurements

of the performance are conservative because of the feature of time-sharing. The real speedup would always be better if the entire machine were available for a single job execution.

In parallel computing, efficiency is perhaps the primary concern. But from a practical point of view, it is of some interest to compare algorithms in terms of computing speeds (costs), e.g. a fast sequential algorithm compared with an efficient parallel algorithm. The cost of a finite element problem is dependent on the solution strategy which is generally categorized as either direct method or iterative method. When using a direct method, the total computations can be anticipated before computing while the cost of an iterative method is dependent upon the rate of convergence and the desired accuracy. Generally, an iterative method can be improved by introducing some parallelism. Thus comparisons are difficult. The parallel algorithm adopted in our research is an iterative method which achieved an almost perfect speedup for the example shown in Table 1 which was solved in 50.45 seconds (including idle time) by 3 processors. The problem was also solved by the conventional finite element method using the skyline solver (Cholesky decomposition) and using 1 processor, and took only 49.27 seconds to solve the entire problem with 10000 elements. At a first examination of the computing time, it would appear pointless to use the parallel algorithm with more processors taking more time. However, a more careful study reveals the following. Besides the computing time, the accuracy desired has to be considered. The maximal relative residual occurring in the skyline solver was 0.017 while the solution obtained from the parallel algorithm was controlled by minimizing the relative residuals to be less than 0.000125.

The parallel algorithm can obtain the solution in 18.82 seconds with the maximal relative residual less than 0.0125 which is even more accurate than the solution obtained from the skyline solver. This shows that the parallel algorithm is actually far superior in this example. Thus for combined computing speed and accuracy, the parallel algorithm can be highly effective.

References

[1] J.-C. Luo, Parallel Algorithms for the Finite Element Method, Ph. D. Thesis, Columbia University, 1988.

[2] J.-C. Luo and M. B. Friedman, Parallel Algorithms for the Finite Element Method, Mechanics Research Communications, Vol. 16 (1), pp. 3-18, 1989.

[3] J.-C. Luo and M. B. Friedman, A Parallel Computational Model for the Finite Element Method on a Memory-Sharing Multiprocessor Computer, Computer Methods in Applied Mechanics and Engineering, (to apperar).

[4] J.-C. Luo and M. B. Friedman, Implicit Decomposition Methods as a Tool for Solving Large-Scale Structural Systems in a Parallel Environment, Computers and Mathematics and Applications, (to apperar in 1990).

[5] J.-C. Luo and M. B. Friedman, A Study On Decomposition Methods, SIAM Scientific and Statistical Computing, (to appear).

[6] J.-C. Luo and M. B. Friedman, On Decomposition Procedures for Reducing the Bandwidth and Profile of a Sparse Matrix, SIAM Scientific and Statistical Computing, (to appear).

[7] J.-C. Luo and M. B. Friedman, New Parallel Algorithms for Finite Element Method on a IBM 3090 Multiprocessor, Journal for Supercomputing (to apperar).

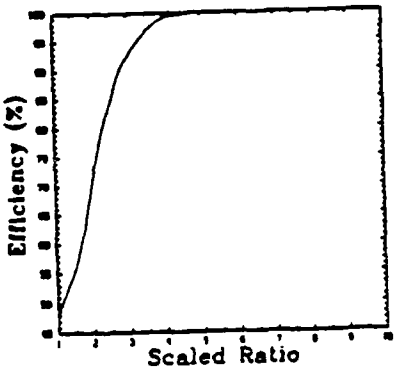

Fig. 1

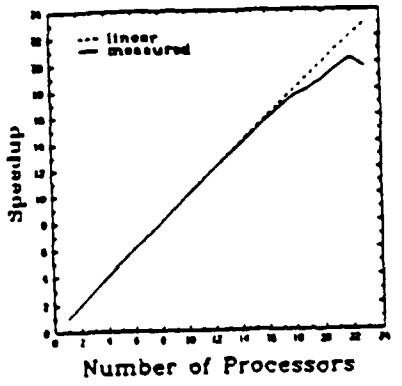

Fig. 2

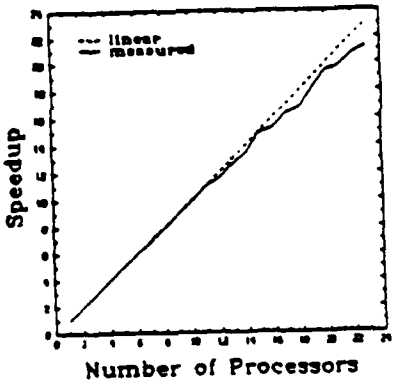

Fig. 3

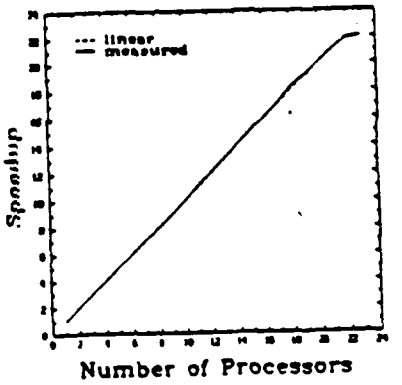

Fig. 4

Figure 5

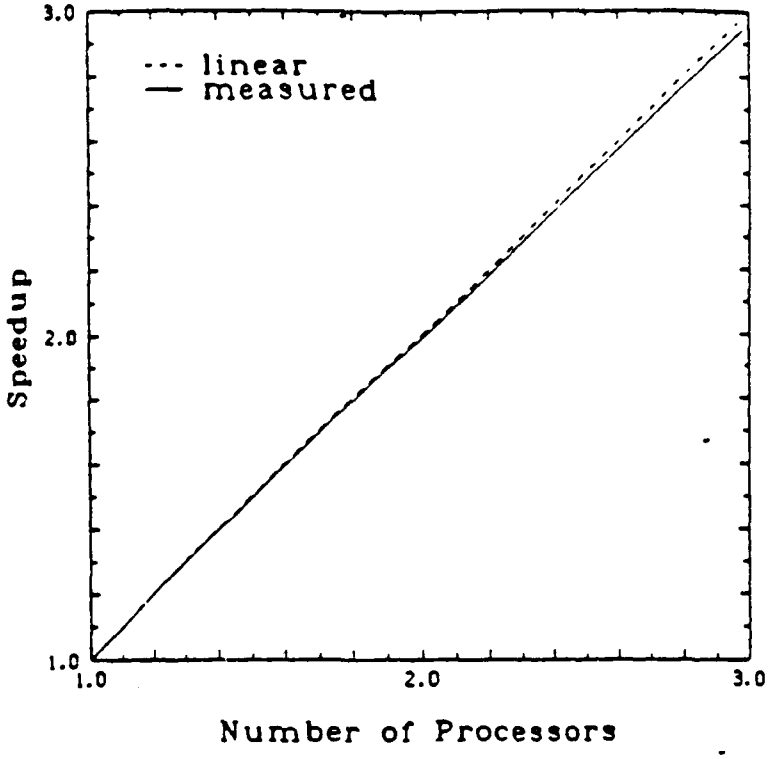

Figure 6